CPSC 330
Applied Machine Learning

Lecture 4: k-Nearest Neighbours and SVM RBFs

UBC 2022-23

Instructor: Mathias Lécuyer

> If two things are similar, the thought of one will tend to trigger the thought of the other
> -- Aristotle

Imports

In [1]:
```
import sys

import IPython
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from IPython.display import HTML

sys.path.append("../code/.")

import ipywidgets as widgets
import mglearn
from IPython.display import display
from ipywidgets import interact, interactive
from plotting_functions import *
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import cross_validate, train_test_split
from utils import *

%matplotlib inline

pd.set_option("display.max_colwidth", 200)
import warnings

warnings.filterwarnings("ignore")
```

Learning outcomes

From this lecture, you will be able to

- explain the notion of similarity-based algorithms;
- broadly describe how k-NNs use distances;
- discuss the effect of using a small/large value of the hyperparameter k when using the k-NN algorithm;
- describe the problem of curse of dimensionality;
- explain the general idea of SVMs with RBF kernel;
- broadly describe the relation of `gamma` and `C` hyperparameters of SVMs with the fundamental tradeoff.

If you want to run this notebook you will have to install `ipywidgets` . Follow the installation instructions here (https://ipywidgets.readthedocs.io/en/latest/user_install.html).

Motivation and distances [video (https://youtu.be/hCa3EXEUmQk)]

Analogy-based models

- Suppose you are given the following training examples with corresponding labels and are asked to label a given test example.

source (https://vipl.ict.ac.cn/en/database.php)

- An intuitive way to classify the test example is by finding the most "similar" example(s) from the training set and using that label for the test example.

Analogy-based algorithms in practice

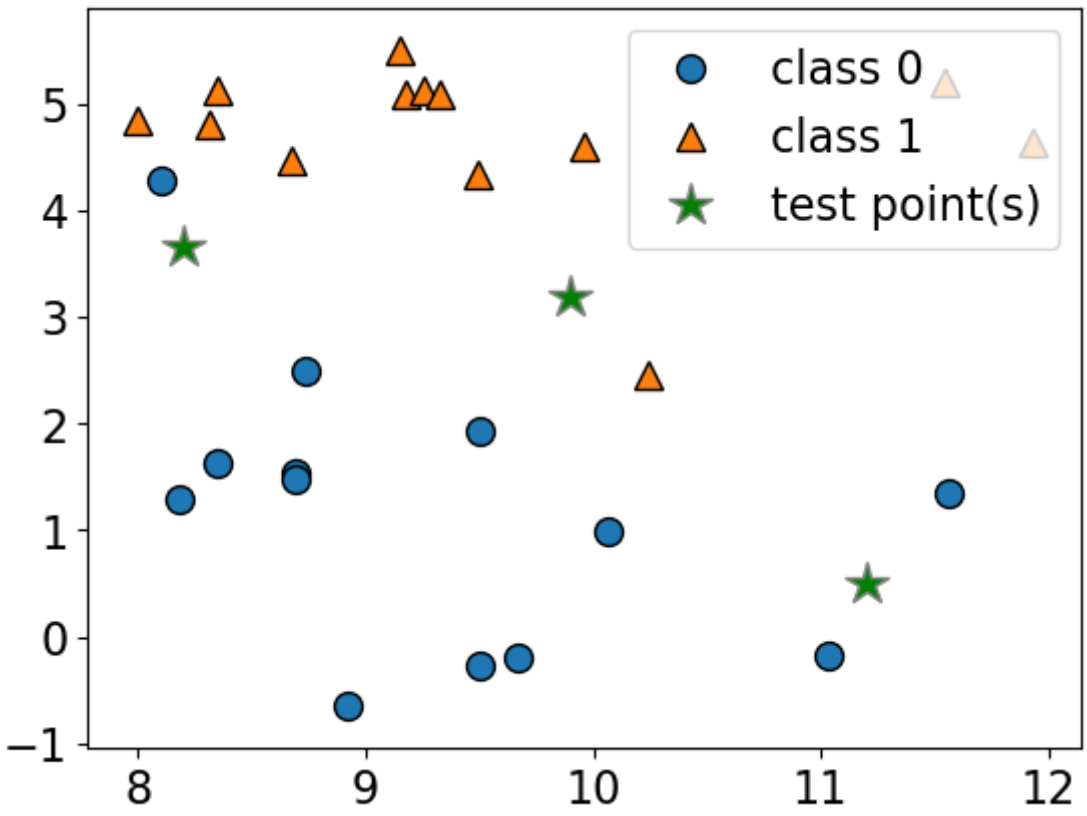
- Herta's High-tech Facial Recognition (https://www.hertasecurity.com/en)
  - Feature vectors for human faces
  - k-NN to identify which face is on their watch list
- Recommendation systems

General idea of k-nearest neighbours algorithm

- Consider the following toy dataset with two classes.
  - blue circles $\rightarrow$ class 0
  - red triangles $\rightarrow$ class 1
  - green stars $\rightarrow$ test examples

In [2]:
```
X, y = mglearn.datasets.make_forge()
X_test = np.array([[8.2, 3.66214339], [9.9, 3.2], [11.2, 0.5]])
```

In [3]:
```
plot_train_test_points(X, y, X_test)
```

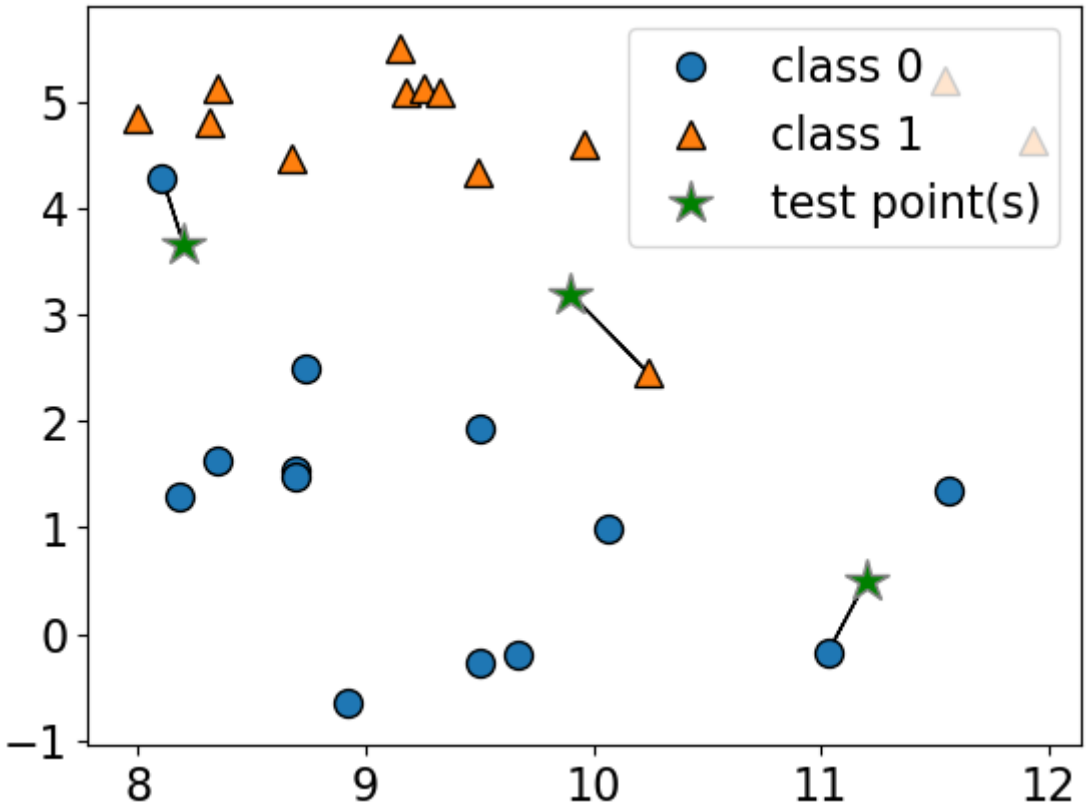


- Given a new data point, predict the class of the data point by finding the "closest" data point in the training set, i.e., by finding its "nearest neighbour" or majority vote of nearest neighbours.

In [4]:
```
def f(n_neighbors):
    return plot_knn_clf(X, y, X_test, n_neighbors=n_neighbors)
```

In [5]:
```
interactive(
    f,
    n_neighbors=widgets.IntSlider(min=1, max=7, step=2, value=1),
)
```

```
n_neighbors 1
```



Out[5]:
```
interactive(children=(IntSlider(value=1, description='n_neighbors', max=
7, min=1, step=2), Output()), _dom_cla…
```

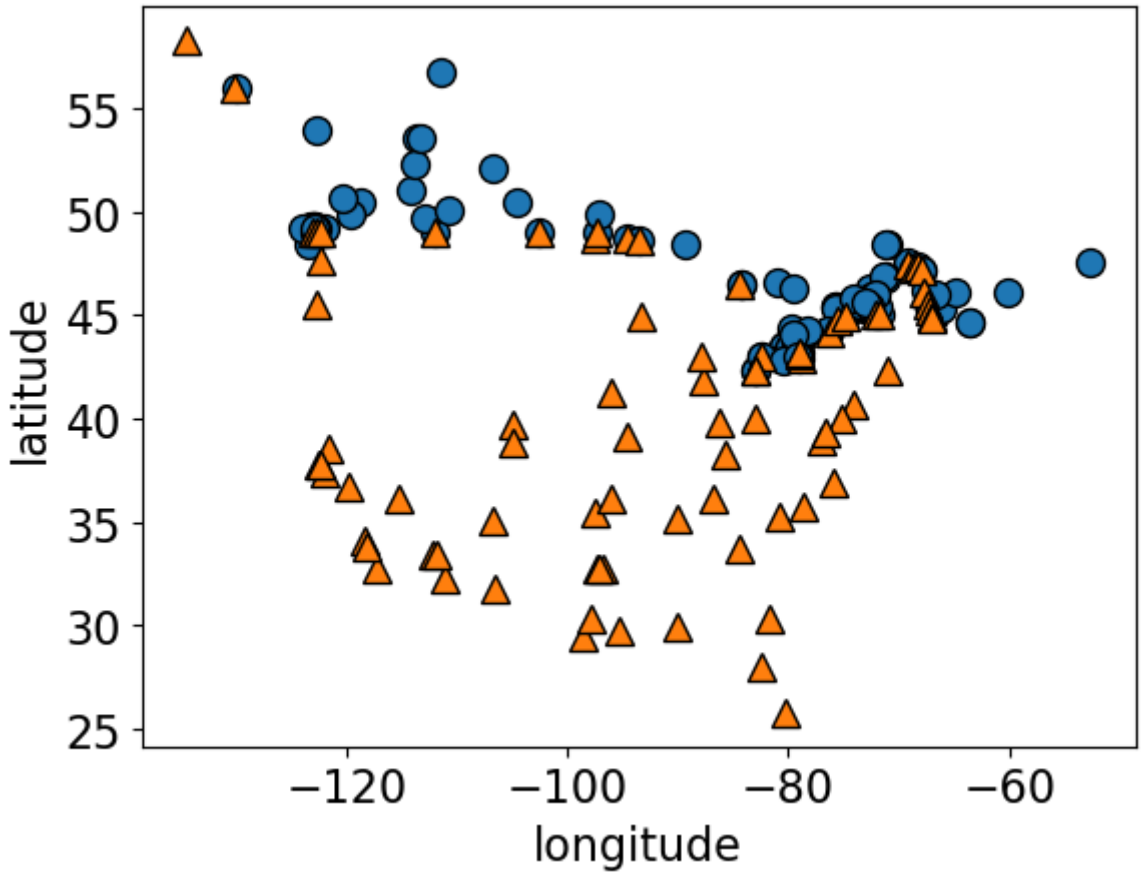
Geometric view of tabular data and dimensions

- To understand analogy-based algorithms it's useful to think of data as points in a high dimensional space.
- Our `X` represents the the problem in terms of relevant **features** (d) with one dimension for each **feature** (column).
- Examples are **points in a d-dimensional space**.

How many dimensions (features) are there in the cities data?

In [7]:
```
cities_df = pd.read_csv("../data/canada_usa_cities.csv")
X_cities = cities_df[["longitude", "latitude"]]
y_cities = cities_df["country"]
```

In [8]:
```
mglearn.discrete_scatter(X_cities.iloc[:, 0], X_cities.iloc[:, 1], y_ci
plt.xlabel("longitude")
plt.ylabel("latitude");
```



- Recall the Spotify Song Attributes (https://www.kaggle.com/geomack/spotifyclassification/home) dataset from homework 1.
- How many dimensions (features) we used in the homework?

In [9]:
```
spotify_df = pd.read_csv("../data/spotify.csv", index_col=0)
X_spotify = spotify_df.drop(columns=["target", "song_title", "artist"])
print("The number of features in the Spotify dataset: %d" % X_spotify.s
X_spotify.head()
```

```
The number of features in the Spotify dataset: 13
```

Out[9]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode
0	0.0102	0.833	204600	0.434	0.021900	2	0.1650	-8.795	1
1	0.1990	0.743	326933	0.359	0.006110	1	0.1370	-10.401	1
2	0.0344	0.838	185707	0.412	0.000234	2	0.1590	-7.148	1
3	0.6040	0.494	199413	0.338	0.510000	5	0.0922	-15.236	1
4	0.1800	0.678	392893	0.561	0.512000	5	0.4390	-11.648	0

Dimensions in ML problems

In ML, usually we deal with high dimensional problems where examples are hard to visualize.

- $d \approx 20$ is considered low dimensional
- $d \approx 1000$ is considered medium dimensional
- $d \approx 100,000$ is considered high dimensional

Feature vectors

Feature vector : is composed of feature values associated with an example.

Some example feature vectors are shown below.

In [10]:
```
print(
    "An example feature vector from the cities dataset: %s"
    % (X_cities.iloc[0].to_numpy())
)
print(
    "An example feature vector from the Spotify dataset: \n%s"
    % (X_spotify.iloc[0].to_numpy())
)
```

```
An example feature vector from the cities dataset: [-130.0437   55.9773]
An example feature vector from the Spotify dataset:
[ 1.02000e-02  8.33000e-01  2.04600e+05  4.34000e-01  2.19000e-02
  2.00000e+00  1.65000e-01 -8.79500e+00  1.00000e+00  4.31000e-01
  1.50062e+02  4.00000e+00  2.86000e-01]
```

Similarity between examples

Let's take 2 points (two feature vectors) from the cities dataset.

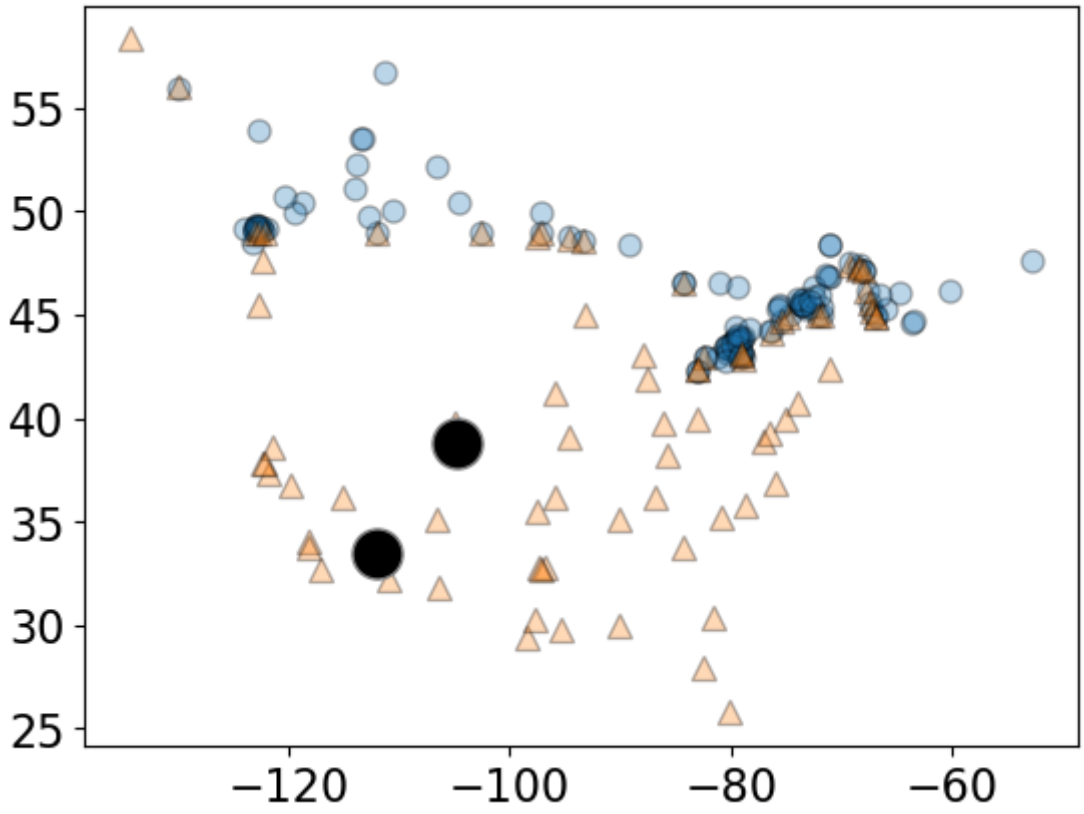
In [11]:
```
two_cities = X_cities.sample(2, random_state=120)
two_cities
```

Out[11]:

	longitude	latitude
69	-104.8253	38.8340
35	-112.0741	33.4484

The two sampled points are shown as big black circles.

In [12]:
```
mglearn.discrete_scatter(
    X_cities.iloc[:, 0], X_cities.iloc[:, 1], y_cities, s=8, alpha=0.3
)
mglearn.discrete_scatter(
    two_cities.iloc[:, 0], two_cities.iloc[:, 1], markers="o", c="k", s
);
```



Distance between feature vectors

- For the cities at the two big circles, what is the *distance* between them?
- A common way to calculate the distance between vectors is calculating the **Euclidean distance**.
- The euclidean distance between vectors $u = <u_1, u_2, \dots, u_n>$ and $v = <v_1, v_2, \dots, v_n>$ is defined as:

$$distance(u, v) = \sqrt{\sum_{i =1}^{n} (u_i - v_i)^2}$$

Euclidean distance

In [13]:
```
two_cities
```

Out[13]:

	longitude	latitude
69	-104.8253	38.8340
35	-112.0741	33.4484

- Subtract the two cities
- Square the difference
- Sum them up
- Take the square root

In [14]:
```
# Subtract the two cities
print("Subtract the cities: \n%s\n" % (two_cities.iloc[1] - two_cities.

# Squared sum of the difference
print(
    "Sum of squares: %0.4f" % (np.sum((two_cities.iloc[1] - two_cities.
)

# Take the square root
print(
    "Euclidean distance between cities: %0.4f"
    % (np.sqrt(np.sum((two_cities.iloc[1] - two_cities.iloc[0]) ** 2)))
)
```

```
Subtract the cities:
longitude   -7.2488
latitude    -5.3856
dtype: float64

Sum of squares: 81.5498
Euclidean distance between cities: 9.0305
```

In [15]:
```
two_cities
```

Out[15]:

	longitude	latitude
69	-104.8253	38.8340
35	-112.0741	33.4484

In [16]:
```
# Euclidean distance using sklearn
from sklearn.metrics.pairwise import euclidean_distances

euclidean_distances(two_cities)
```

Out[16]:
```
array([[0.        , 9.03049217],
       [9.03049217, 0.        ]])
```

Note: `scikit-learn` supports a number of other distance metrics (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html).

Finding the nearest neighbour

- Let's look at distances from all cities to all other cities

In [17]:
```
dists = euclidean_distances(X_cities)
np.fill_diagonal(dists, np.inf)
print("All distances: %s\n\n%s" % (dists.shape, dists))
```

```
All distances: (209, 209)

[[        inf  4.95511263  9.869531   ... 52.42640992 58.03345923
  51.49856241]
 [ 4.95511263         inf 14.6775792  ... 57.25372435 62.77196948
  56.25216034]
 [ 9.869531   14.6775792          inf ... 44.23515175 50.24972011
  43.69922405]
 ...
 [52.42640992 57.25372435 44.23515175 ...         inf  6.83784786
   3.32275537]
 [58.03345923 62.77196948 50.24972011 ...  6.83784786         inf
   6.55573969]
 [51.49856241 56.25216034 43.69922405 ...  3.32275537  6.55573969
          inf]]
```

Let's look at the distances between City 0 and some other cities.

In [18]:
```
print("Feature vector for city 0: \n%s\n" % (X_cities.iloc[0]))
print("Distances from city 0 to the first 5 cities: %s" % (dists[0][:5]
# We can find the closest city with `np.argmin`:
print(
    "The closest city from city 0 is: %d \n\nwith feature vector: \n%s"
    % (np.argmin(dists[0]), X_cities.iloc[np.argmin(dists[0])])
)
```

```
Feature vector for city 0:
longitude   -130.0437
latitude      55.9773
Name: 0, dtype: float64

Distances from city 0 to the first 5 cities: [        inf  4.95511263  9.
869531   10.10645223 10.44966612]
The closest city from city 0 is: 81

with feature vector:
longitude   -129.9912
latitude      55.9383
Name: 81, dtype: float64
```

Ok, so the closest city to City 0 is City 81.

Question

- Why did we set the diagonal entries to infinity before finding the closest city?

Finding the distances to a query point

We can also find the distances to a new "test" or "query" city:

In [19]:
```
# Let's find a city that's closest to the a query city
query_point = [[-80, 25]]

dists = euclidean_distances(X_cities, query_point)
dists[0:10]
```

Out[19]:
```
array([[58.85545875],
       [63.80062924],
       [49.30530902],
       [49.01473536],
       [48.60495488],
       [39.96834506],
       [32.92852376],
       [29.53520104],
       [29.52881619],
       [27.84679073]])
```

In [20]:
```
# The query point is closest to
print(
    "The query point %s is closest to the city with index %d and the di
    % (query_point, np.argmin(dists), dists[np.argmin(dists)])
)
```

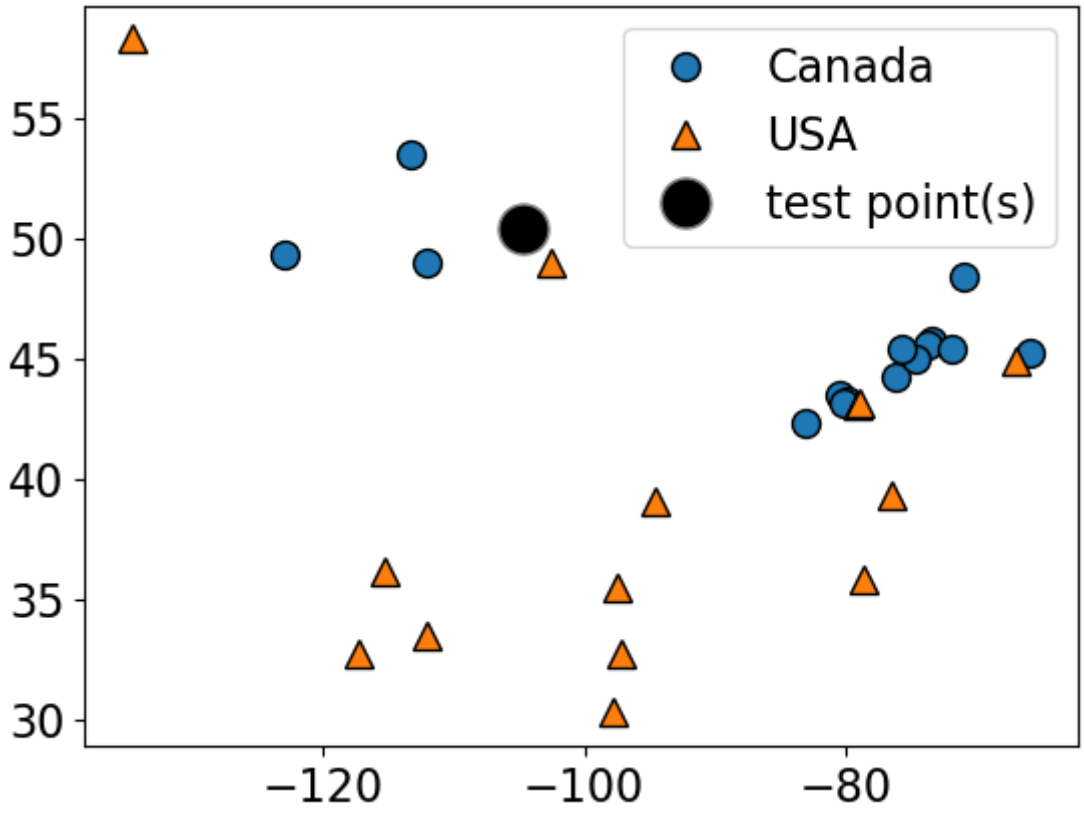
```
The query point [[-80, 25]] is closest to the city with index 72 and the
distance between them is: 0.7982
```

k-Nearest Neighbours (k-NNs) [video (https://youtu.be/bENDqXKJLmg)]

In [21]:
```
small_cities = cities_df.sample(30, random_state=90)
one_city = small_cities.sample(1, random_state=44)
small_train_df = pd.concat([small_cities, one_city]).drop_duplicates(ke
```

In [22]:
```
X_small_cities = small_train_df.drop(columns=["country"]).to_numpy()
y_small_cities = small_train_df["country"].to_numpy()
test_point = one_city[["longitude", "latitude"]].to_numpy()
```

In [23]:
```
plot_train_test_points(
    X_small_cities,
    y_small_cities,
    test_point,
    class_names=["Canada", "USA"],
    test_format="circle",
)
```



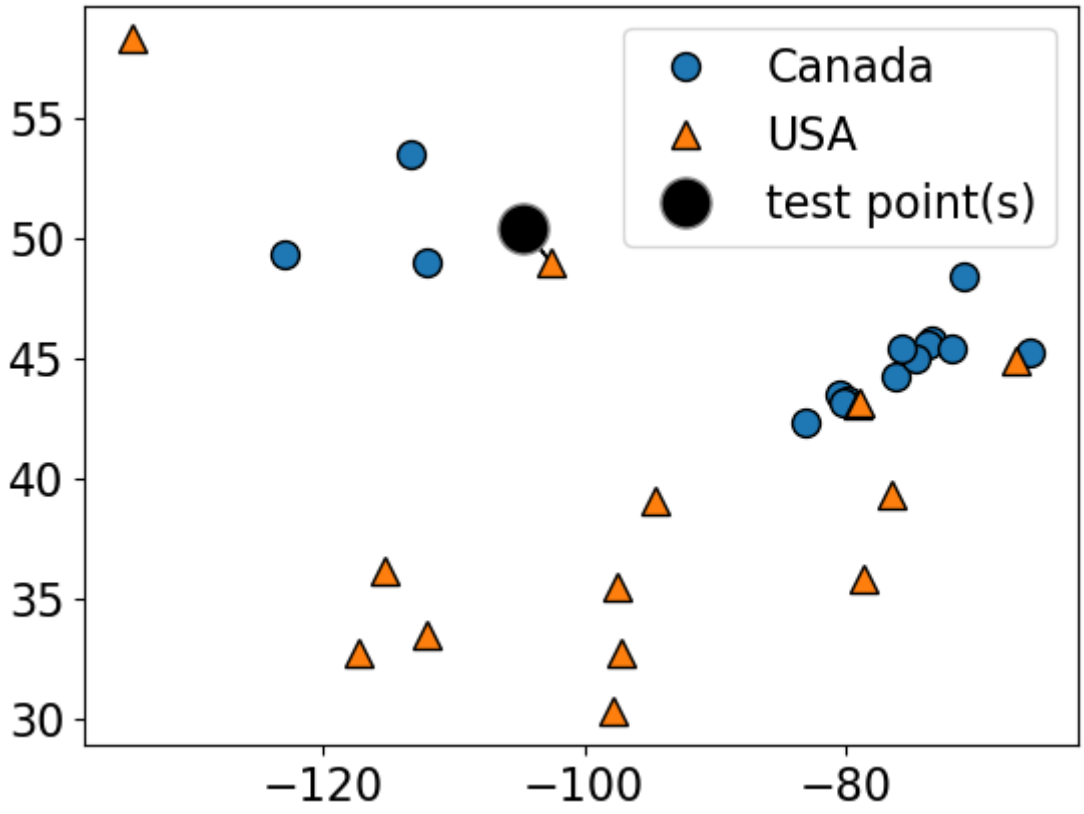
- Given a new data point, predict the class of the data point by finding the "closest" data point in the training set, i.e., by finding its "nearest neighbour" or majority vote of nearest neighbours.

Suppose we want to predict the class of the black point.

- An intuitive way to do this is predict the same label as the "closest" point ($k = 1$) (1-nearest neighbour)
- We would predict a target of **USA** in this case.

In [24]:
```
plot_knn_clf(
    X_small_cities,
    y_small_cities,
    test_point,
    n_neighbors=1,
    class_names=["Canada", "USA"],
    test_format="circle",
)
```

```
n_neighbors 1
```

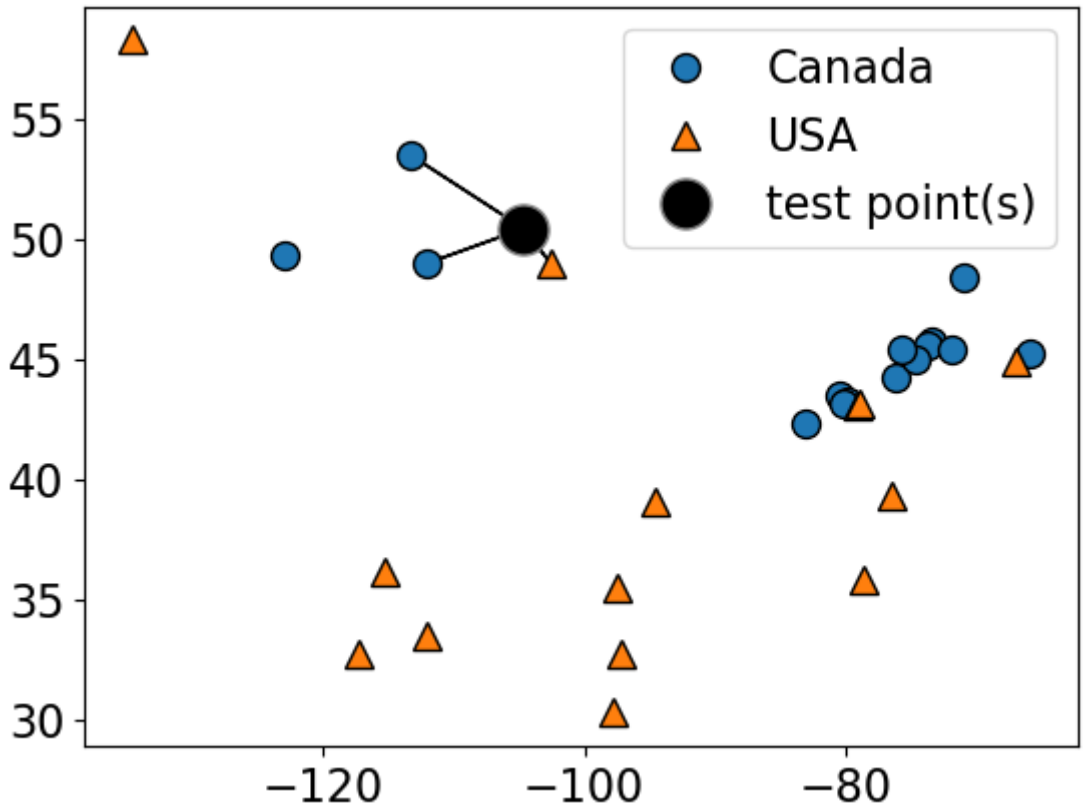


How about using $k > 1$ to get a more robust estimate?

- For example, we could also use the 3 closest points ($k = 3$) and let them **vote** on the correct class.
- The **Canada** class would win in this case.

In [25]:
```
plot_knn_clf(
    X_small_cities,
    y_small_cities,
    test_point,
    n_neighbors=3,
    class_names=["Canada", "USA"],
    test_format="circle",
)
```

```
n_neighbors 3
```



In [26]:
```
from sklearn.neighbors import KNeighborsClassifier

k_values = [1, 3]

for k in k_values:
    neigh = KNeighborsClassifier(n_neighbors=k)
    neigh.fit(X_small_cities, y_small_cities)
    print(
        "Prediction of the black dot with %d neighbours: %s"
        % (k, neigh.predict(test_point))
    )
```

```
Prediction of the black dot with 1 neighbours: ['USA']
Prediction of the black dot with 3 neighbours: ['Canada']
```

Question

- Is it a good or a bad idea to consider an odd number for k? Why or why not?

Choosing `n_neighbors`

- The primary hyperparameter of the model is `n_neighbors` (k) which decides how many neighbours should vote during prediction?
- What happens when we play around with `n_neighbors` ?
- Are we more likely to overfit with a low `n_neighbors` or a high `n_neighbors` ?
- Let's examine the effect of the hyperparameter on our cities data.

In [27]:
```
X = cities_df.drop(columns=["country"])
y = cities_df["country"]

# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1, random_state=123
)
```

In [28]:
```
k = 1
knn1 = KNeighborsClassifier(n_neighbors=k)
scores = cross_validate(knn1, X_train, y_train, return_train_score=True
pd.DataFrame(scores)
```

Out[28]:

	fit_time	score_time	test_score	train_score
0	0.003854	0.006703	0.710526	1.0
1	0.001873	0.002923	0.684211	1.0
2	0.001859	0.002859	0.842105	1.0
3	0.001891	0.003029	0.702703	1.0
4	0.001831	0.002857	0.837838	1.0

In [29]:
```
k = 100
knn100 = KNeighborsClassifier(n_neighbors=k)
scores = cross_validate(knn100, X_train, y_train, return_train_score=Tr
pd.DataFrame(scores)
```

Out[29]:

	fit_time	score_time	test_score	train_score
0	0.002043	0.141483	0.605263	0.600000
1	0.001216	0.002506	0.605263	0.600000
2	0.001294	0.002617	0.605263	0.600000
3	0.001195	0.002623	0.594595	0.602649
4	0.001272	0.002432	0.594595	0.602649

In [30]:
```
def f(n_neighbors=1):
    results = {}
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_validate(knn, X_train, y_train, return_train_score=T
    results["n_neighbours"] = [n_neighbors]
    results["mean_train_score"] = [round(scores["train_score"].mean(),
    results["mean_valid_score"] = [round(scores["test_score"].mean(), 3
    print(pd.DataFrame(results))


interactive(
    f,
    n_neighbors=widgets.IntSlider(min=1, max=101, step=10, value=1),
)
```

Out[30]:
```
interactive(children=(IntSlider(value=1, description='n_neighbors', max=1
01, min=1, step=10), Output()), _dom_…
```

In [1]:
```
plot_knn_decision_boundaries(X_train, y_train, k_values=[1, 11, 100])
```

```
---------------------------------------------------------------------------
--
NameError                                 Traceback (most recent call las
t)
Cell In[1], line 1
----> 1 plot_knn_decision_boundaries(X_train, y_train, k_values=[1, 11, 1
00])

NameError: name 'plot_knn_decision_boundaries' is not defined
```

How to choose `n_neighbors` ?

- `n_neighbors` is a hyperparameter
- We can use hyperparameter optimization to choose `n_neighbors` .

In [32]:
```
results_dict = {
    "n_neighbors": [],
    "mean_train_score": [],
    "mean_cv_score": [],
    "std_cv_score": [],
    "std_train_score": [],
}
param_grid = {"n_neighbors": np.arange(1, 50, 5)}

for k in param_grid["n_neighbors"]:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_validate(knn, X_train, y_train, return_train_score=T
    results_dict["n_neighbors"].append(k)

    results_dict["mean_cv_score"].append(np.mean(scores["test_score"]))
    results_dict["mean_train_score"].append(np.mean(scores["train_score
    results_dict["std_cv_score"].append(scores["test_score"].std())
    results_dict["std_train_score"].append(scores["train_score"].std())

results_df = pd.DataFrame(results_dict)
```

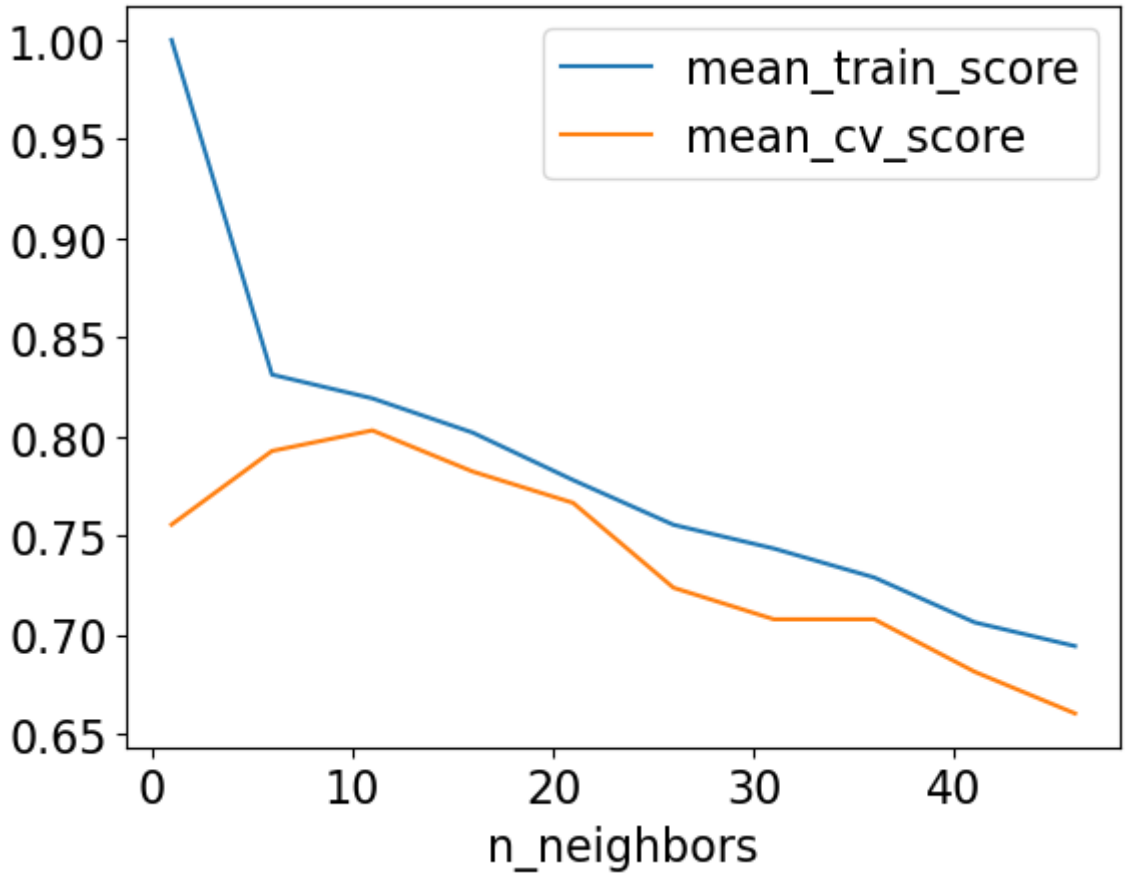
In [33]:
```
results_df = results_df.set_index("n_neighbors")
results_df
```

Out[33]:

n_neighbors	mean_train_score	mean_cv_score	std_cv_score	std_train_score
1	1.000000	0.755477	0.069530	0.000000
6	0.831135	0.792603	0.046020	0.013433
11	0.819152	0.802987	0.041129	0.011336
16	0.801863	0.782219	0.074141	0.008735
21	0.777934	0.766430	0.062792	0.016944
26	0.755364	0.723613	0.061937	0.025910
31	0.743391	0.707681	0.057646	0.030408
36	0.728777	0.707681	0.064452	0.021305
41	0.706128	0.681223	0.061241	0.018310
46	0.694155	0.660171	0.093390	0.018178

In [34]:
```
1 results_df[["mean_train_score", "mean_cv_score"]].plot()
```

Out[34]: <AxesSubplot: xlabel='n_neighbors'>



In [35]:
```
1 best_n_neighbours = results_df.idxmax()["mean_cv_score"]
2 best_n_neighbours
```

Out[35]: 11

Let's try our best model on test data.

In [36]:
```
1 knn = KNeighborsClassifier(n_neighbors=best_n_neighbours)
2 knn.fit(X_train, y_train)
3 print("Test accuracy: %0.3f" % (knn.score(X_test, y_test)))
```

```
Test accuracy: 0.905
```

1. Analogy-based models find examples from the test set that are most similar to the query example we are predicting.
2. A dataset with 10 dimensions is considered low dimensional.
3. Euclidean distance will always have a positive value.

? ? Questions on distances and k-NNs

Exercise 4.1

What would be the Euclidean distance between the following two vectors `u` and `v` ?

In [41]:
```
u = np.array([0, 0, 20, -2])
v = np.array([-1, 0, 18, -4])

np.sum((u-v)*(u-v))
```

Out[41]: 9

Exercise 4.2 k-NN practice questions

1. When we calculated Euclidean distances from all cities to all other cities, why did we set the diagonal entries to infinity before finding the closest city?
2. Consider this toy dataset:

$$ X = \begin{bmatrix}5 & 2\\4 & 3\\ 2 & 2\\ 10 & 10\\ 9 & -1\\ 9& 9\end{bmatrix}, \quad y = \begin{bmatrix}0\\0\\1\\1\\1\\2\end{bmatrix}.$$

- If $k=1$, what would you predict for $x=\begin{bmatrix} 0\\0\end{bmatrix}$?
- If $k=3$, what would you predict for $x=\begin{bmatrix} 0\\0\end{bmatrix}$?

Recap - k-NNs

At home, you watched videos on analogy-based models, in particular k-nearest neighbors.

Let's answer some questions about what you have learned (T/F):

1. Unlike with decision trees, with k-NNs most of the work is done at the `predict` stage.
2. With k-NN, setting the hyperparameter k to larger values typically reduces training error.
3. Similar to decision trees, k-NNs finds a small set of good features.
4. In k-NN, the classification of the closest neighbour to the test example always contributes the most to the prediction.

Open questions

- Is it a good or a bad idea to consider an odd number for k ? Why or why not?
- In the video, diagonal entries of the Euclidian distance matrix were changed from 0 to infinity. Why?
- Why is the training error always 0.0 when k = 1?

More on k-NNs [video (https://youtu.be/IRGbqi5S9gQ)]

Other useful arguments of `KNeighborsClassifier`

- `weights` $\rightarrow$ When predicting label, you can assign higher weight to the examples which are closer to the query example.
- Exercise for you: Play around with this argument. Do you get a better validation score?

(Optional) Regression with k-nearest neighbours (k-NNs)

- Can we solve regression problems with k-nearest neighbours algorithm?
- In k-NN regression we take the average of the k-nearest neighbours.
- We can also have weighted regression.

See an example of regression in the lecture notes.

In [ ]:
```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

In [ ]:
```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

Pros of k-NNs for supervised learning

- Easy to understand, interpret.
- Simple hyperparameter k (`n_neighbors`) controlling the fundamental tradeoff.
- Can learn very complex functions given enough data.
- Lazy learning: Takes no time to `fit`

Cons of k-NNs for supervised learning

- Can be potentially be VERY slow during prediction time, especially when the training set is very large.
- Often not that great test accuracy compared to the modern approaches.
- It does not work well on datasets with many features or where most feature values are 0 most of the time (sparse datasets).

For regular k-NN for supervised learning (not with sparse matrices), you should scale your features. We'll be looking into it soon.

Parametric vs non parametric

- You might see a lot of definitions of these terms.
- A simple way to think about this is:
  - do you need to store at least $O(n)$ worth of stuff to make predictions? If so, it's non-parametric.
- Non-parametric example: k-NN is a classic example of non-parametric models.
- Parametric example: decision stump
- If you want to know more about this, find some reading material here (https://www.cs.ubc.ca/~schmidtm/Courses/340-F16/L6.pdf), here (http://mlss.tuebingen.mpg.de/2015/slides/ghahramani/gp-neural-nets15.pdf), and here (https://machinelearningmastery.com/parametric-and-nonparametric-machine-learning-algorithms/).
- By the way, the terms "parametric" and "non-paramteric" are often used differently by statisticians, see here (https://help.xlstat.com/s/article/what-is-the-difference-between-a-parametric-and-a-nonparametric-test?language=en_US) for more...

$\mathcal{O}(n)$ is referred to as big $\mathcal{O}$ notation. It tells you how fast an algorithm is or how much storage space it requires. For example, in simple terms, if you have n examples and you need to store them all you can say that the algorithm requires $\mathcal{O}(n)$ worth of stuff.

Curse of dimensionality

- Affects all learners but especially bad for nearest-neighbour.
- k-NN usually works well when the number of dimensions d is small but things fall apart quickly as d goes up.
- If there are many irrelevant attributes, k-NN is hopelessly confused because all of them contribute to finding similarity between examples.
- With enough irrelevant attributes the accidental similarity swamps out meaningful similarity and k-NN is no better than random guessing.

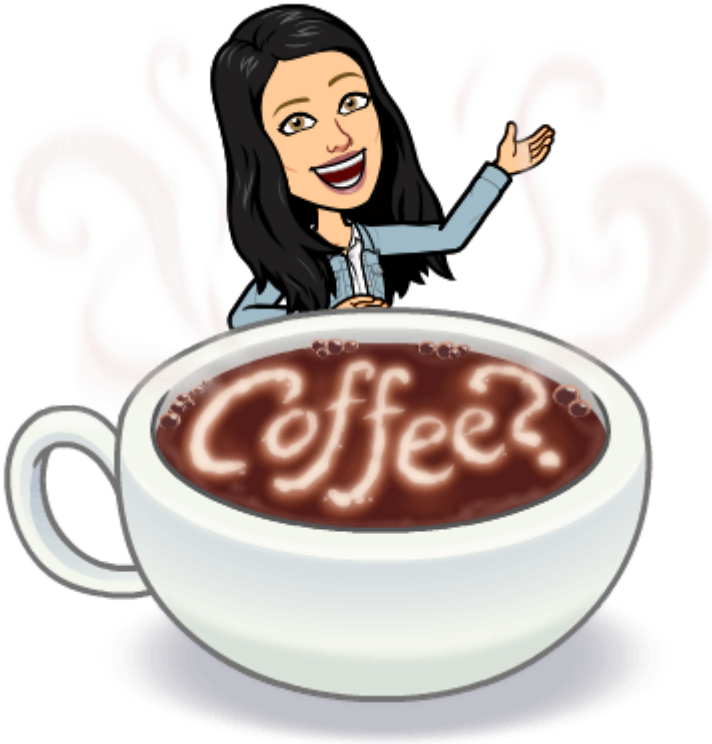
In [42]:
```
from sklearn.datasets import make_classification

nfeats_accuracy = {"nfeats": [], "dummy_valid_accuracy": [], "KNN_valid
for n_feats in range(4, 2000, 100):
    X, y = make_classification(n_samples=2000, n_features=n_feats, n_cl
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=123
    )
    dummy = DummyClassifier(strategy="most_frequent")
    dummy_scores = cross_validate(dummy, X_train, y_train, return_train

    knn = KNeighborsClassifier()
    scores = cross_validate(knn, X_train, y_train, return_train_score=T
    nfeats_accuracy["nfeats"].append(n_feats)
    nfeats_accuracy["KNN_valid_accuracy"].append(np.mean(scores["test_s
    nfeats_accuracy["dummy_valid_accuracy"].append(np.mean(dummy_scores
```

In [43]:
```
pd.DataFrame(nfeats_accuracy)
```

Out[43]:

	nfeats	dummy_valid_accuracy	KNN_valid_accuracy
0	4	0.510000	0.980000
1	104	0.502500	0.716250
2	204	0.501875	0.761250
3	304	0.503125	0.713750
4	404	0.508750	0.669375
5	504	0.500625	0.651250
6	604	0.505000	0.655000
7	704	0.505625	0.641250
8	804	0.506250	0.636875
9	904	0.500000	0.620000
10	1004	0.508750	0.610000
11	1104	0.503750	0.631875
12	1204	0.502500	0.600000
13	1304	0.504375	0.557500
14	1404	0.510625	0.568125
15	1504	0.503750	0.572500
16	1604	0.499375	0.576250
17	1704	0.503125	0.599375
18	1804	0.508125	0.551875
19	1904	0.506875	0.590625

Break (5 min)



Support Vector Machines (SVMs) with RBF kernel [video (https://youtu.be/ic_zqOhi020)]

- Very high-level overview
- Our goals here are
  - Use `scikit-learn` 's SVM model.
  - Broadly explain the notion of support vectors.
  - Broadly explain the similarities and differences between k-NNs and SVM RBFs.
  - Explain how `C` and `gamma` hyperparameters control the fundamental tradeoff.

> (Optional) RBF stands for radial basis functions. We won't go into what it means in this class. Refer to this video (https://www.youtube.com/watch?v=Qc5IyLW_hns) if you want to know more.

Overview

- Another popular similarity-based algorithm is Support Vector Machines with RBF Kernel (SVM RBFs)

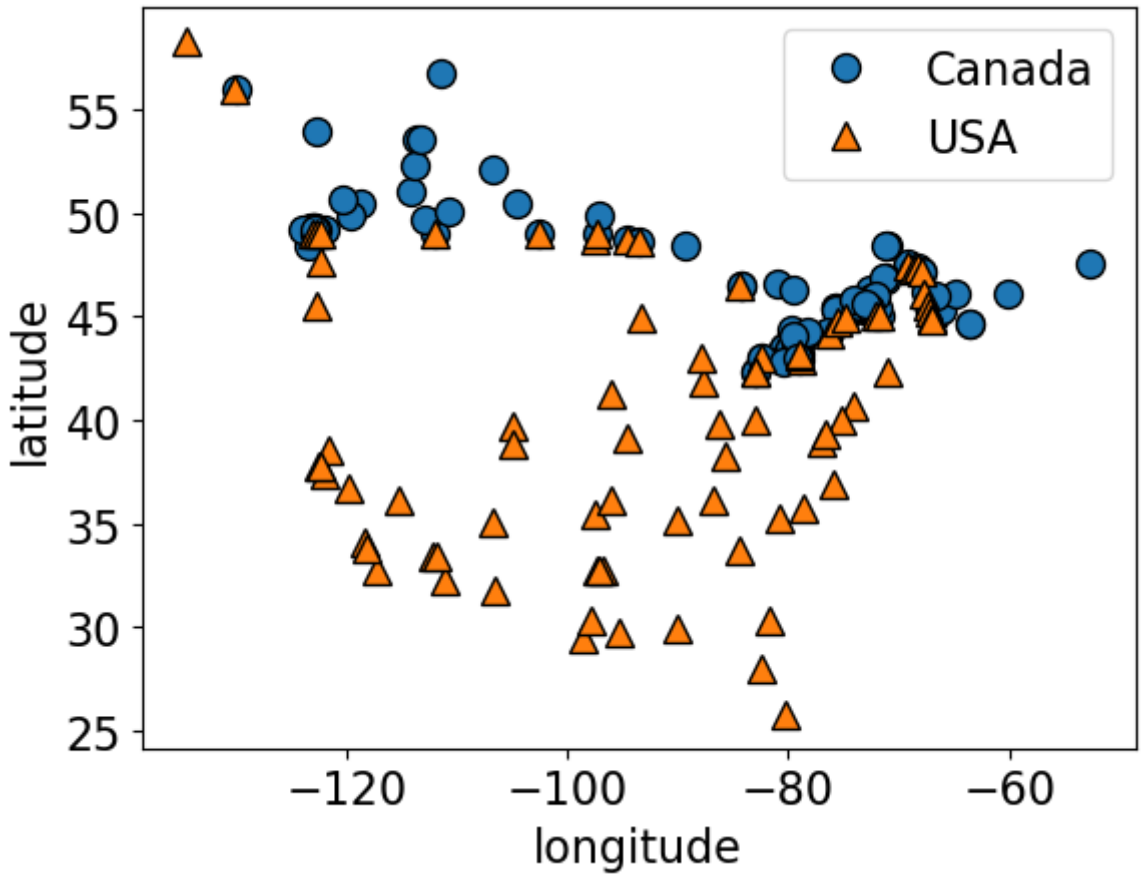
- Superficially, SVM RBFs are more like weighted k-NNs.
  - The decision boundary is defined by **a set of positive and negative examples** and **their weights** together with **their similarity measure**.
  - A test example is labeled positive if on average it looks more like positive examples than the negative examples.

- The primary difference between k-NNs and SVM RBFs is that
  - Unlike k-NNs, SVM RBFs only remember the key examples (support vectors). So it's more efficient than k-NN.
  - SVMs use a different similarity metric which is called a "kernel" in SVM land. A popular kernel is Radial Basis Functions (RBFs)
  - They usually perform better than k-NNs!

Let's explore SVM RBFs

Let's try SVMs on the cities dataset.

In [44]:
```
mglearn.discrete_scatter(X_cities.iloc[:, 0], X_cities.iloc[:, 1], y_ci
plt.xlabel("longitude")
plt.ylabel("latitude")
plt.legend(loc=1);
```



In [45]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X_cities, y_cities, test_size=0.2, random_state=123
)
```

In [46]:
```
knn = KNeighborsClassifier(n_neighbors=best_n_neighbours)
scores = cross_validate(knn, X_train, y_train, return_train_score=True)
print("Mean validation score %0.3f" % (np.mean(scores["test_score"])))
pd.DataFrame(scores)
```

```
Mean validation score 0.803
```

Out[46]:

	fit_time	score_time	test_score	train_score
0	0.005274	0.006239	0.794118	0.819549
1	0.002166	0.003047	0.764706	0.819549
2	0.001877	0.003529	0.727273	0.850746
3	0.001637	0.002546	0.787879	0.828358
4	0.001719	0.002608	0.939394	0.783582

In [47]:
```
from sklearn.svm import SVC

svm = SVC(gamma=0.01)  # Ignore gamma for now
scores = cross_validate(svm, X_train, y_train, return_train_score=True)
print("Mean validation score %0.3f" % (np.mean(scores["test_score"])))
pd.DataFrame(scores)
```

```
Mean validation score 0.820
```

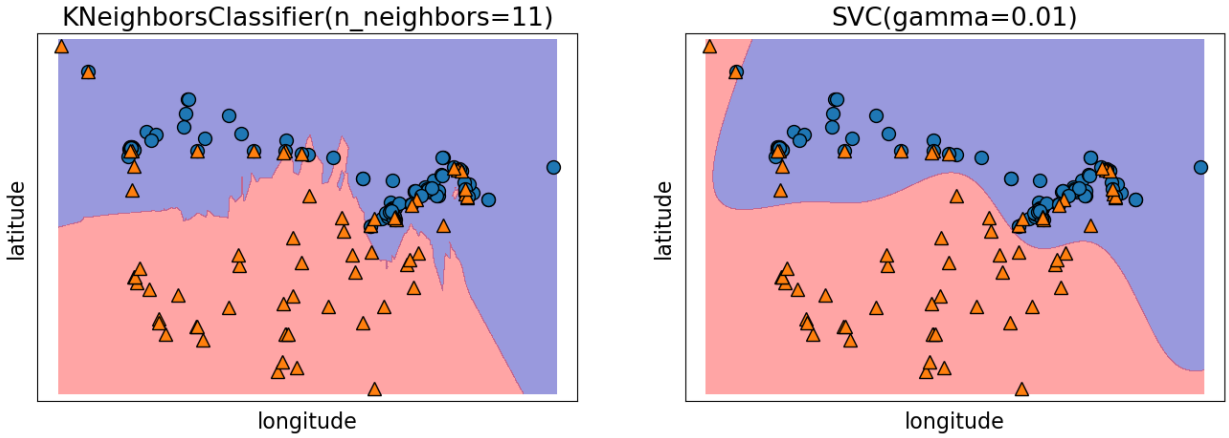
Out[47]:

	fit_time	score_time	test_score	train_score
0	0.006933	0.002283	0.823529	0.842105
1	0.002420	0.001686	0.823529	0.842105
2	0.002157	0.001490	0.727273	0.858209
3	0.002399	0.001730	0.787879	0.843284
4	0.002109	0.001452	0.939394	0.805970

Decision boundary of SVMs

- We can think of SVM with RBF kernel as "smooth KNN".

In [48]:
```
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

for clf, ax in zip([knn, svm], axes):
    clf.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(
        clf, X_train.to_numpy(), fill=True, eps=0.5, ax=ax, alpha=0.4
    )
    mglearn.discrete_scatter(X_train.iloc[:, 0], X_train.iloc[:, 1], y_
    ax.set_title(clf)
    ax.set_xlabel("longitude")
    ax.set_ylabel("latitude")
```

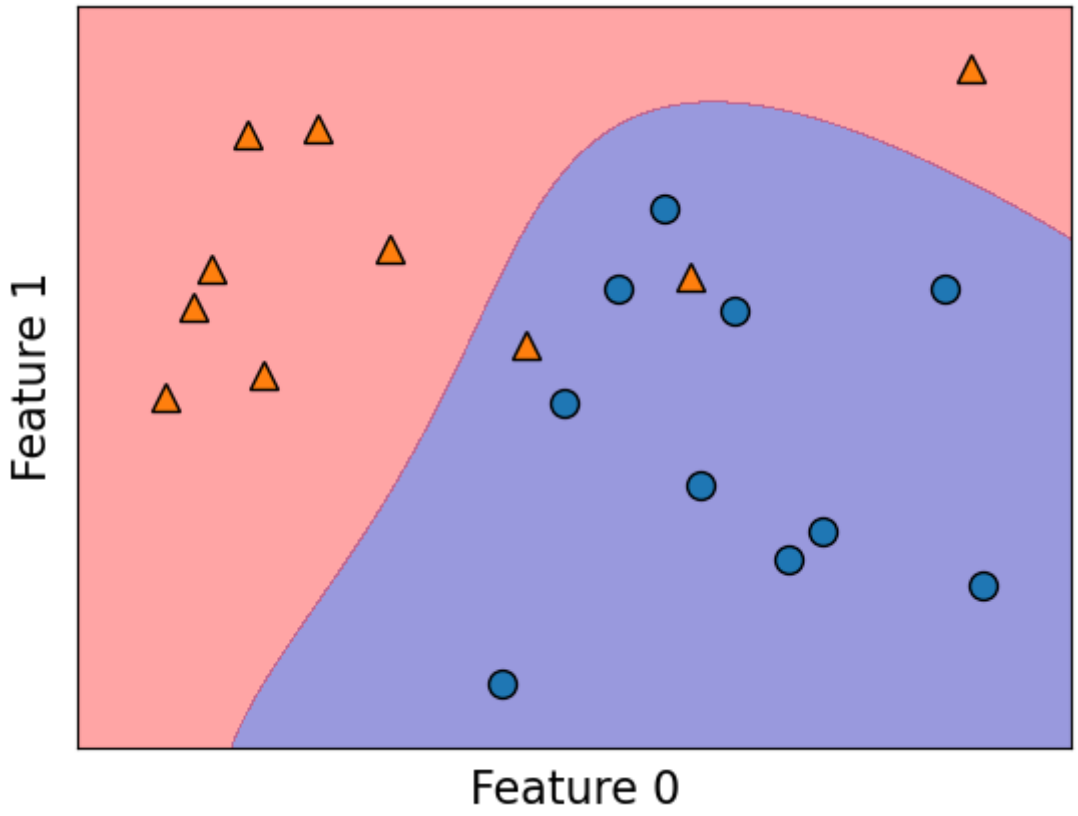


Support vectors

- Each training example either is or isn't a "support vector".
  - This gets decided during `fit`.
- **Main insight: the decision boundary only depends on the support vectors.**
- Let's look at the support vectors.

In [49]:
```
from sklearn.datasets import make_blobs

n = 20
n_classes = 2
X_toy, y_toy = make_blobs(
    n_samples=n, centers=n_classes, random_state=300
)  # Let's generate some fake data
```

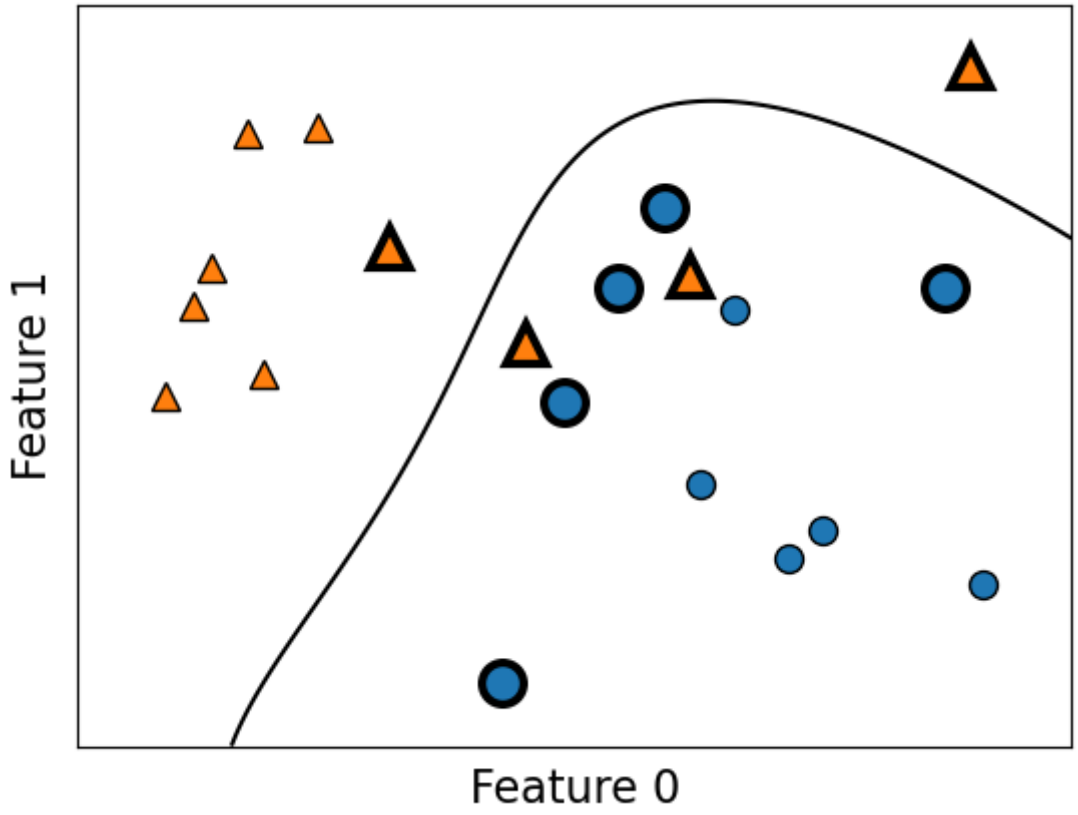
In [50]:
```
mglearn.discrete_scatter(X_toy[:, 0], X_toy[:, 1], y_toy)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
svm = SVC(kernel="rbf", C=10, gamma=0.1).fit(X_toy, y_toy)
mglearn.plots.plot_2d_separator(svm, X_toy, fill=True, eps=0.5, alpha=0
```



In [51]:
```
svm.support_
```

Out[51]:
```
array([ 3,  8,  9, 14, 19,  1,  4,  6, 17], dtype=int32)
```

In [52]:
```
1 plot_support_vectors(svm, X_toy, y_toy)
```



The support vectors are the bigger points in the plot above.

Hyperparameters of SVM

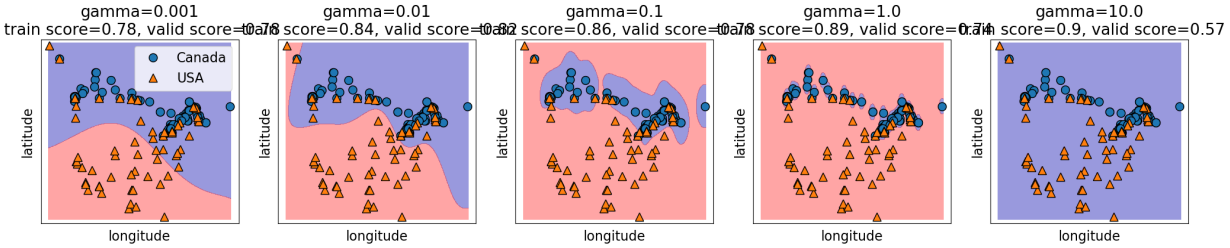
- Key hyperparameters of `rbf` SVM are
  - `gamma`
  - `C`
- We are not equipped to understand the meaning of these parameters at this point but you are expected to describe their relation to the fundamental tradeoff.

See `scikit-learn` 's explanation of RBF SVM parameters (https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html).

Relation of `gamma` and the fundamental trade-off

- `gamma` controls the complexity (fundamental trade-off), just like other hyperparameters we've seen.
  - larger `gamma` $\rightarrow$ more complex
  - smaller `gamma` $\rightarrow$ less complex

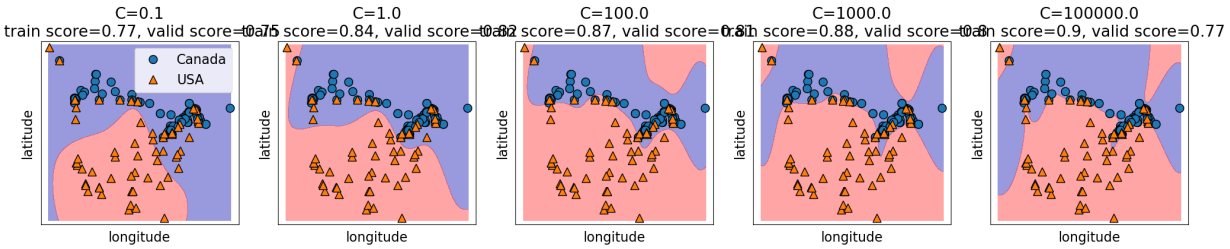
In [53]:
```
gamma = [0.001, 0.01, 0.1, 1.0, 10.0]
plot_svc_gamma(
    gamma,
    X_train.to_numpy(),
    y_train.to_numpy(),
    x_label="longitude",
    y_label="latitude",
)
```



Relation of `C` and the fundamental trade-off

- `C` *also* affects the fundamental tradeoff
  - larger `C` $\rightarrow$ more complex
  - smaller `C` $\rightarrow$ less complex

In [54]:
```
C = [0.1, 1.0, 100.0, 1000.0, 100000.0]
plot_svc_C(
    C, X_train.to_numpy(), y_train.to_numpy(), x_label="longitude", y_l
)
```



Search over multiple hyperparameters

- So far you have seen how to carry out search over a hyperparameter
- In the above case the best training error is achieved by the most complex model (large `gamma`, large `C`).
- Best validation error requires a hyperparameter search to balance the fundamental tradeoff.
  - In general we can't search them one at a time.
  - More on this in the Hyperparameter optimization lecture. But if you cannot wait till then, you may look up the following:
    - sklearn.model_selection.GridSearchCV (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)
    - sklearn.model_selection.RandomizedSearchCV (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.htr

SVM Regressor

- Similar to KNNs, you can use SVMs for regression problems as well.
- See `sklearn.svm.SVR` (https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html) for more details.

❓ ❓ Questions on SVM RBFs

Case study

A friend of yours shows you a SVM model they are working on, with very high accuracy on the training set (close to 100%).

Upon closer inspection, you find out that almost all the original samples have been selected as support vectors.

What can you guess about the values of `C` and `gamma` used to train this model?

Do you think this model will generalize well?

More practice questions

- Check out some more practice questions here (https://ml-learn.mds.ubc.ca/en/module4).

Summary

- We have KNNs and SVMs as new supervised learning techniques in our toolbox.
- These are analogy-based learners and the idea is to assign nearby points the same label.
- Unlike decision trees, all features are equally important.
- Both can be used for classification or regression (much like the other methods we've seen).

Coming up:

Lingering questions:

- Are we ready to do machine learning on real-world datasets?
- What would happen if we use k-NNs or SVM RBFs on the spotify dataset from hw2?
- What happens if we have missing values in our data?
- What do we do if we have features with categories or string values?

In [ ]:
```
1
```